Completeness of a Fact Extractor

Yuan Lin, Richard C. Holt, Andrew J. Malton
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, ON N2L 3G1, Canada
{y3lin,holt,ajmalton}@waterloo.ca

Abstract

The process of software reverse engineering commonly uses an extractor, which parses source code and extracts facts about the code. The level of detail in these facts varies from extractor to extractor. This paper describes four levels of increasingly detailed completeness of these facts: (semantic completeness, compiler completeness, syntax completeness and source completeness) and introduces the concept of relative completeness of extractors. Validating that an extractor correctly produces facts at a given level of completeness is in general very challenging. This paper gives a method for validating the semantic completeness of an extractor, and describes the application of this method to CPPX, an extractor for C or C++ based on GCC.

1. Introduction

Software reverse engineering extracts and presents information about existing software systems. A key part of this activity is automated by fact extractors which input source code and produce facts about the code. These facts can be thought of as rows in a relational data base table, or as edges in a graph. For example, the fact *(call, P, Q)*, could mean that function *P* calls function *Q*.

The authors have been involved in developing of a number of fact extractors, most recently CPPX (C++ Extractor) [8]. This extractor is based on the GCC open source C++ compiler [13]. The GCC front end transforms the source program into a corresponding Abstract Syntax Graph (ASG); see Figure 1.

The ASG is an abstraction the program's syntax tree decorated with edges that correspond to resolution of references to declarations and with attributes representing information such as line numbers.

Ordinarily, the back end of GCC proceeds to translate the ASG to assembly language. CPPX operates by replacing GCC's back end by a graph transformer, called cppx (written in lower case to emphasize that it is only part of the CPPX fact extractor). The cppx transformation produces another version of the ASG, which satisfies the Datrix schema [5] for representing facts about C or C++ programs. The Datrix schema is designed to be convenient for reverse engineering purposes.

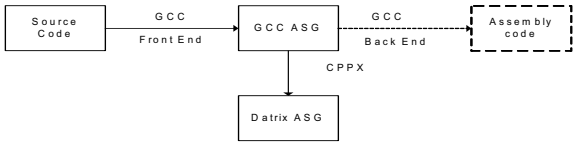

Figure 1. CPPX fact extractor. CPPX consists of the GCC front end together with the CPPX graph transformer, shown here as boxes and arrows drawn with solid lines

There has been considerable research on validating fact extractors [18] [4] [19] [6]. In this paper we record the approach we used toward validating the CPPX extractor. More generally, we give a method for validating that an extractor is semantically complete, that is, that its extracted facts contain enough information to recover a program with the same behavior as the original source program.

The rest of this paper is organized as follows. Section 2 provides details about compiler phases and ASGs. Section 3 provides background information on factbases, schemas and exchange formats. Section 4 defines four increasingly detailed levels of facts that an extractor may produce and introduces the concept of relative

Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)
1095-1350/03 $ 17.00 © 2003 IEEE

completenss of extractors. Section 5 gives a method for validating the semantic completeness of a fact extractor and explains how we applied this method to CPPX. Section 6 discusses the cppx graph transformer. Section 7 lists problems in CPPX that were detected as a result of applying our validation method. Finally, section 8 summarizes the research and proposes future work.

2. Compiler Phases and ASGs

In a typical compiler [1][3], the source code is initially preprocessed, parsed, and subjected to semantic analysis. In the case of GCC, these steps are carried out by its front end; see Figure 1. Following these initial steps, the assembly or machine code is generated, optimized and emitted. These final steps are carried out by the compiler's back end.

The result of parsing is a parse tree. Lexical details of the source, such as spacing, comments, preprocessor directives, do not appear in the tree. This tree is then simplified and made more convenient for further processing. The simplified tree is called the Abstract Syntax Tree (AST).

The semantic analysis phase of compilation decorates the AST by adding semantic information such as types of identifiers, declaration locations and overload resolution. The decorated AST is called the Abstract Semantic Graph (ASG). After the insertion of semantic information, the syntactic structure may be simplified further:

The nodes of the ASG represent source program entities including types, classes, methods, statements, expressions, and so on down to the lowest level of constants and variable references. The edges represent relationships between them. There are two kinds of edges in the ASG: tree edges and semantic edges.

Tree edges give the tree structure of the ASG. They represent containment in the source syntax. For example, a declaration is contained by its scope, a declared identifier by its declaration, a variable reference by an expression involving it, and a conditional expression by the *if* statement which it controls.

Semantic edges (non-tree edges) represent semantic connections, such as typing and the resolution of scoped names. For example, semantic edges connect the operands of an expression to their declarations and an instance declaration to its class type.

3. Schemas and Interchange Formats

An extractor such as CPPX can be used for a variety of reverse engineering purposes, so its output (the Datrix ASG factbase) should be available in a well documented, accessible *format*. The format should determine a concrete syntax, so the ASG can be conveniently input by further reverse engineering tools such as visualizers and analyzers.

A number of exchange formats have been proposed for exchange of reverse engineering factbases, including GXL [20] (based on XML), TA [14], RSF [21], and Grax [10]. CPPX generates TA by default, with an option to emit GXL.

Besides determining a concrete syntax, exchange formats such as TA and GXL allow the user to specify a *schema* or data model. The schema constrains the relationships between facts in the factbase and is used to give an interpretation of those facts. For example, the schema may constrain *Call* edges to connect only *Function* nodes, and its interpretation may indicate that each call in the source code is to have a corresponding *Call* edge in the factbase. The key benefit of supporting schemas, and not just a fixed exchanged format, is that this allows the exchange format to be broadly used, across a set of applications. The user of the format creates a special schema to handle the data of interest, such as the data generated by CPPX.

Examples of fact exchange schemas include the Datrix schema [5], the Columbus schema [11] and the Dagstuhl Middle-Level Model [17]. CPPX uses the Datrix schema. (Technically speaking, CPPX uses a slightly modified version of the Datrix schema. To simplify the presentation in this paper, we will refer to these both as simply the "Datrix schema".)

4. Completeness of Fact Extractors

An extractor is analogous to a compiler in that it inputs source code and translates it to data in a very different form. In the case of a compiler, the target data is assembly or machine language, for use in linking and execution. In the case of an extractor, the target data is facts about the source, for use in reverse engineering.

The extracted factbase may include only high level information, such as interactions between global entities such as functions and classes, or may also contain detailed information down to the level of statements and expressions.

IEEE COMPUTER SOCIETY

Table 1. Four levels of completeness for a fact extractor

Level	Question	Definition
1. Source complete	Are original and recovered source programs p_0 and p_1 identical, byte for byte, including comments and spacing?	$p_1 = R(g_0)$
2. Syntax complete	Are original and recovered syntax trees t_0 and t_1 identical?	$t_0 = FE(p_0)$ $t_1 = FE(p_1)$
3. Compiler complete	Are original and recovered assembly code a_0 and a_1 identical?	$a_0 = BE(t_0)$ $a_1 = BE(t_1)$
4. Semantically complete	Are original and recovered behaviors s_0 and s_1 equivalent?	$s_0=Sem(p_0)$ $s_1=Sem(p_1)$

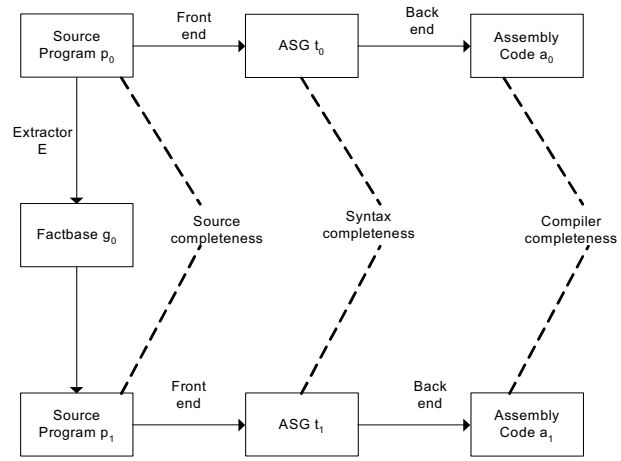

Figure 2. Validating three kinds of completeness

4.1. Four Levels of Completeness

It is useful to characterize the extracted factbase in terms of how *complete* it is. At the most inclusive extreme, the factbase can be *source complete*, meaning that it is possible to recover the exact source program, byte for byte, including comments and white space, from the factbase. Most extractors including CPPX are not source complete, because information such as white space is not usually needed and would bloat the factbase with unwanted detail.

A fact extractor E inputs the original source program p_0 and produces factbase g_0, i.e., $g_0 = E(p_0)$; see Table 1. In this figure, FE and BE are the Front End and Back End of a compiler. Sem maps a program to its semantics

In Table 1 we define four levels of completeness for an extractor. These levels are a generalization of source completeness as defined by Dean et al [8]. At each level, completeness is defined by whether the extracted factbase retains enough information to answer a certain question. Figure 2 illustrates how the questions in levels 1 to 3 might be addressed by means of testing.

4.2. Hierarchy of Completeness

From top to bottom in Table 1, or from left to right in Figure 2, completeness becomes weaker, i.e., the levels form a completeness hierarchy in which less information needs to be retained in the factbase as the level number increases.

If an extractor is source complete, it is also syntax complete, because the extracted factbase g_0 can be transformed back to the original source program, from which the original syntax tree can be derived. By a similar argument, source completeness also implies compiler completeness and semantic completeness.

Syntax completeness implies both compiler completeness and semantic completeness because the syntax tree incorporates all the information about a program that is needed for code generation. Assuming the compiler is correct, compiler completeness implies semantic completeness because if original source and the recovered source have identical generated assembly language then they must have the same semantics.

We have left the definition of syntax completeness somewhat ambiguous, allowing it to be based on either the context free parse tree or on the ASG, whichever is most convenient for a given purpose.

4.3. Semantic Completeness

The lowest level in Table 1 is semantic completeness, which retains information about program behavior. This level is particularly interesting because this information is needed by many reverse engineering tasks. This is the level of completeness that CPPX (supposedly) attains.

In level 4 of Table 1, we have assumed there is a semantics function *Sem*, which maps a source program to a representation of its behavior. Unfortunately, for production languages such as C and C++, such a function

IEEE COMPUTER SOCIETY

has not been formally defined. Furthermore, the equality of semantics is undecidable: In general we can't check if two programs p_0 and p_1 have the same behavior. Since semantic completeness asks whether p_0 and p_1 are semantically equivalent, it seems that determining if an extractor is semantically complete must be quite a challenge! We take up that challenge in the last half of this paper.

Many extractors do not extract enough information to satisfy any of the levels in Table 1. For example, the CFX extractor [12] only extracts information at the level of functions and global variables along with the interactions among them. This level is sufficient for certain reverse engineering analyses such as recovery of architecture, but is not sufficient for analysis involving function bodies. (We might call this "architecture complete" if we assume that architectural design recovery is based only on functions and global variables.)

CPPX's goal was to retain as much useful information as possible from the GCC ASG, and more information than is retained by most existing fact extractors. As a result, we decided to try to make CPPX to semantically complete.

4.4. Relative Completeness

While the four kinds of completeness we have described are important, they are not the only possible kinds of extractor completeness. Consider the possibility that a *use-def* graph [2] is needed to detect dead code. We define an extractor to be *use-def complete* if its created factbase contains enough information to produce a use-def graph.

More generally, we will now introduce the formal concept of *relative completeness* among translators and extractors, as follows. Suppose a source program p is translated by transformation T to produce T(p). For example, T might be the front end FE of a compiler or might be an extractor for *use-def* graphs. Program p is also transformed by extractor E to produce information E(p). We define that E is *at least as complete as* T if the information produced by E can be further processed to create the information created by T. (See Figure 3.) Formally, we define this as follows:

Definition 1. If there exists a function F such that for all p

$$F(E(p)) = T(p)$$

then we say E is *at least as complete as* T. We write this as:

$$E \geq_c T$$

See Figure 3 for illustration of T, E and F.

Although Definition 1 is intuitively appealing, in actual practice, as illustrated by Figure 2, we used the following definition involving a recovery transformation R:

Definition 2. If there exists a function R such that for all p

$$T(R(E(p))) = T(p)$$

then we say E is *at least as complete as* T. We write this as:

$$E \geq_c T$$

This second definition states that E is more complete than T if E's output E(p) can be recovered back to R(E(p)) which T processes to output equivalent to T(p). Fortunately, the two definitions are equivalent, as we will now show.

Proposition. Definitions 1 and 2 are equivalent.

We will now prove this proposition. It is obvious that if R exists, F also exists, because F can be defined in terms of R as

F(e) = T(R(e)).

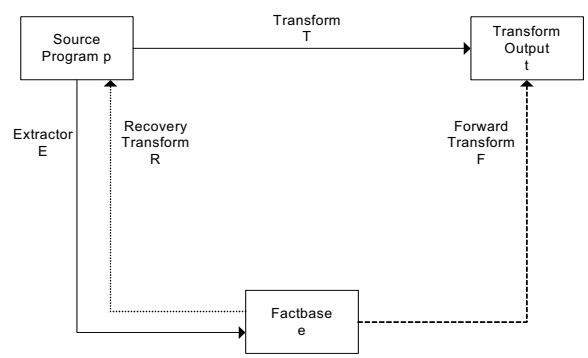

Figure 3. Relative completeness of E and T

Now consider the converse: suppose F exists such that F(E(p)) = T(p). We show that R exists such that T(R(E(p)) = T(p). Suppose T1(t) is a left inverse of T, that is, T1(T(p)) = p. Then let R (e) = T1(F(e)), and we have T(R(E(p)) = T(T1(F(E(p)) = T(T1(T(p)) = T(p) as required. Such T1 exists since it only needs to be defined on the range of T. End of proof.

Aside. Note that if F, E, and T are programs, then R can also be a program: that is, R is computable. Given an extraction e, it is sufficient (for R) to generate programs p in the source language and test them successively until one be found for which T(p) = e. However, this is not very efficient! In practice R examines e, which is a data

IEEE COMPUTER SOCIETY

structure, and constructs p from the information therein.

Although we have explained the concept of relative completeness in terms of extractors and translators, this concept is purely mathematical, and can be applied to any functions. The relative completeness operator $\geq_c$ forms a lattice. Its top element is ID (the identity function):

$$\forall\ T \bullet ID \geq_c T$$

which means that no transform can preserve more information than does the identity function. Its least elements are any constant function K:

$$\forall\ T \bullet T \geq_c K$$

which means that translating the input program to a constant, such as the null string, loses all information about the input.

The four hierarchical levels of completeness (source, syntax, compiler and semantic completeness) can be defined in terms of relative completeness. For example, extractor E is compiler complete for compiler C if $E \geq_c C$. E is ASG complete for front end FE if $E \geq_c FE$. E is source complete for compiler C if $E \geq_c ID$, which means that the facts extracted by E can be used to reconstruct the source program.

With this discussion of completeness behind us, we are ready to explain how we validated CPPX's semantic completeness.

5. Validating CPPX's Semantic Completeness

This section describes the method we used to validate the semantic completeness of CPPX. Perhaps the most obvious approach to validate CPPX is to run it against a large test suite of source programs, and to check that the extracted factbase for each test is correct. This approach would be very expensive, as it requires extensive manual work to do the checking or to create putatively correct factbases for comparison to CPPX generated factbases. As we will explain, our approach to validation of CPPX uses a test suite of programs, but avoids this manual step.

The structure of CPPX was introduced using Figure 1. (Recall that CPPX consists of the front end of GCC together with the cppx graph transformation.)

Figure 4 expands Figure 1 by adding a transformation called RCCPX (Reverse CPPX), which recovers a source program p_1 from CPPX's extracted ASG g_0. The figure shows recovered source p_1 being compiled by GCC's front end and back end to produce assembly code a_1.

If all phases in the diagram are working correctly and CPPX is *compiler complete*, then the assembly language a_0 from any original source p_0 will be the same as its recovered assembly language a_1 from recovered source p_1. As is explained below, our validation method for CPPX works by checking that a_0 and a1 are identical for a suite of test source programs.

5.1. Why are Assembly Codes a_0 and a_1 Identical?

Suppose we run a source program p_0 through the phases shown in Figure 4. If we find that a_0 and a_1 identical, what can we conclude? It is tempting to conclude that this implies that cppx is *semantics complete*. However, this might not be the case. If GCC's back end is severely buggy and acts as a constant function K, producing null output for all input, then a_0 and a_1 will always be identical, regardless of the actions of CPPX. Since GCC is widely used in production, it is clear that its back end is not severely buggy. In fact, we can safely assume that its back end and front end together reliably generate correct assembly output for most input source programs.

As we will explain below, the recovery transformation RCPPX is written to be simple and easy to make correct, so we can expect that it is reasonably reliable. So, we can conclude that if there is a failure in one of the phases shown in Figure 4, the failure will probably occur in the least reliable phase, namely, in the cppx transformation phase.

Even if CPPX is *semantically complete*, it would be wrong to expect a_0 and a_1 to be identical. After all, GCC is an optimizing compiler, and the smallest change in its input, as p_0 varies to p_1, may produce a change from a_0 to a_1 while maintaining the same semantics. So, it may come as a surprise to learn that when we run test source programs though the configuration in Figure 4, a_0 and a_1 turn out to be identical (ignoring the cases when we encounter a bug in CPPX). We will now answer the question: Why are a_0 and a_1 identical?

Although GCC's back end is extremely complex, it is deterministic, i.e., for the same input it always generates the same output assembly code. So, if the GCC ASGs t_0 and t_1, for the top and bottom lines in Figure 2, are the same, then a_0 and a_1 will be identical. So the question becomes: Can we expect the GCC ASGs to be the same for source programs p_0 and p_1?

To our mild disappointment, we discovered that GCC's front end does some low level program transformations.

IEEE COMPUTER SOCIETY

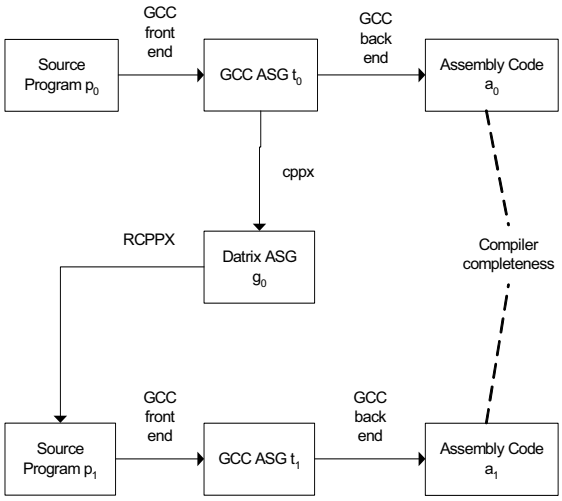

Figure 4. Validating Completeness of CPPX

For example, it rewrites the integer comparison j >= 2 into j > 1. Because of this, one might conclude that it is unlikely that the two GCC ASGs t_0 and t_1 can be identical. It turns out that this conclusion is not warranted for the following reason.

Apparently, the purpose of these rewrites is to produce a standard or canonical form for a few special constructs, and that these rewrites are idempotent, i.e., doing them a second time has no effect. If we denote this rewriting as function f, we can expect or hope that, $f(p_0) = f(p_1)$. What we observed was that, with cppx working properly, the two ASGs were effectively the same, i.e., they were treated the same by the back end. The idempotent property of the GCC front end together with the deterministic nature of the GCC back end yields the overall property that, when cppx is operating correctly, a_0 and a_1 are identical.

After our initial validation of CPPX was largely finished, we ran tests to see if ASGs t_0 and t_1 were the same. We found that they were very similar but not identical. They differed in their information about file and line location. They also differed due to renaming of local variables, as described below. These differences do not effect the generated code. Probably they could be eliminated by deleting file and line information and by optionally disabling the renaming of local variables, but we have not tried this.

In short, CPPX is almost ASG-complete, but our method of validation only shows the technically weaker property of compiler completeness. Compiler completeness seems to be more "surprising" and satisfying as its definition doesn't rely on the internal details of the compiler.

5.2. Suite of Test Programs

In our validation of CPPX, we used a test suite of C and C++ programs derived from two sources. The first part of this suite consists of a set of small programs that were designed to show how source programs are represented when using the Datrix schema. The second part consists of the suite of C and C++ programs used to test the GCC compiler. Combined, these two parts provide an initial reasonable coverage of the features of C and C++. (Note that we have thus far concentrated on validating CPPX on C rather than on C++ input.)

5.3. The Resolution Problem

As we have explained, CPPX generates an ASG in which *tree* edges represent the containment structure and *semantic* edges give connections in the tree. One key use of semantic connections to represent references to declarations. For example, in the following program, there is a reference from the final x on line 4 to the local declaration of x (line 3), not to the global declaration of x (line 1).

```
1 int x;
2 int f() {
3   int x;
4   x++;
5 }
```

In the corresponding ASG, there is an edge from the node for the final x to the node for the declaration of the local x and not to the global declaration. This edge is redundant in the sense that the scope rules of the programming language imply which declaration the final x is referencing. Despite this redundancy, such edges are important because they allow the *resolution problem* (the sometimes difficult problem of resolving appropriate references) to be reliably solved once, in the extractor. Indeed, one of the reasons CPPX is based on GCC is to take advantage of the fact that GCC's ASG provides a reliable solution to this problem.

Now consider the possibility that CPPX might incorrectly solve the resolution problem, for example, by generating the reference edge from the final x to the global x instead of to the local x. Unfortunately, this error would not be detected by our validation method for CPPX, because in the recovered source program p_1, there is no

IEEE COMPUTER SOCIETY

record of this edge and hence no evidence of the error. Although CPPX creates this edge, RCPPX deletes it when recovering the source program.

We devised the following approach to allow our validation method to check that resolution edges in the factbase are correct: local (non-external) identifiers are systematically renamed to be unique, by suffixing additional characters. For the above example, the recovered program p_1 would have this form:

```
1 int x;
2 int f() {
3   int x_2;
4   x_2++;
5 }
```

This renaming is done in way that avoids clashing with any global identifiers which might have a similar form. If the global had been called 'x_2' then this would of course not be chosen as the new name of the local.

With this suffixing convention, if CPPX generates incorrect reference edges, the resulting semantic errors will be reflected in the recovered code. As a result, our validation method effectively checks that reference edges are correct, and that the CPPX has correctly solved the resolution problem.

6. RCPPX: Recovering Source from Facts

As can be seen in Figure 4, to use our validation method, we need a recovery transformation, namely we need RCPPX. RCPPX inputs a Datrix ASG in the TA exchange format and outputs the equivalent C/C++ source program. TA is a relational notation, whose underlying data model is much like a Relational Data Base. In TA, there is a set of triples, recording the source, target and type of each edge in a graph. In this case, the graph is the Datrix ASG. As well, there are triples that represent attributes. In TA, this information is encoded as an ASCII stream, stored in a flat file.

The following is a tiny C compilation unit, which will serve as an example:

```
/* A tiny program */
int x;
void glup (int y) {
  x = y + 1;
  }
```

The CPPX extractor translates this program to TA notation (simplified here for presentation purposes) as follows:

```
FACT TUPLE :
$INSTANCE @0 cFunction
cRefersTo @2 @3
$INSTANCE @2 cNameRef
$INSTANCE @3 cObject
$INSTANCE @4 cFormalObject
$INSTANCE @5 cBlock
$INSTANCE @7 cBuiltInType
cRefersTo @9 @4
$INSTANCE @9 cNameRef
$INSTANCE @10 cBuiltInType
contain @40 @2
$INSTANCE @15 cBuiltInType
cInstance @3 @10
contain @40 @55
contain @55 @9
contain @55 @66
contain @0 @5
cInstance @0 @7
cInstance @4 @10
contain @0 @4
cInstance @66 @10
contain @5 @40
$INSTANCE @40 cOperator
$INSTANCE @55 cOperator
$INSTANCE @66 cLiteral
contain @577 @3
contain @577 @0
$INSTANCE @577 cScopeCompil

FACT ATTRIBUTE :
@0 { name = glup }
@2 { name = x }
@3 { name = x }
@4 { name = y }
@7 { name = void }
@9 { name = y }
@10 { name = int }
@40 { op = asgn-eq }
@55 { op = bplus }
@66 { value = 1 }
```

This TA represents a graph, which is diagrammed in Figure 5. By comparing the source program with its image in TA (or with the diagram in Figure 5), it can be seen that CPPX preserves the structural information, but deletes comments and layout. Syntax in the source code is replaced by explicit relationships in the TA.

It is the job of RCPPX to bridge the gap from TA back to source code. RCPPX does this in three steps, first by a renaming script, second by transforming to nested syntax, and third by two structural transformations implemented in TXL [7].

6.1. Suffixing Script.

This first step modifies program identifiers as explained in section 5C above, in order to validate the extractor's computation of semantic edges. The result in the TA for the sample program is to replace

```
@4 { name = y }
```

by

IEEE COMPUTER SOCIETY

```
@4 { name = y_1 }
```

and similarly for node `@9`.

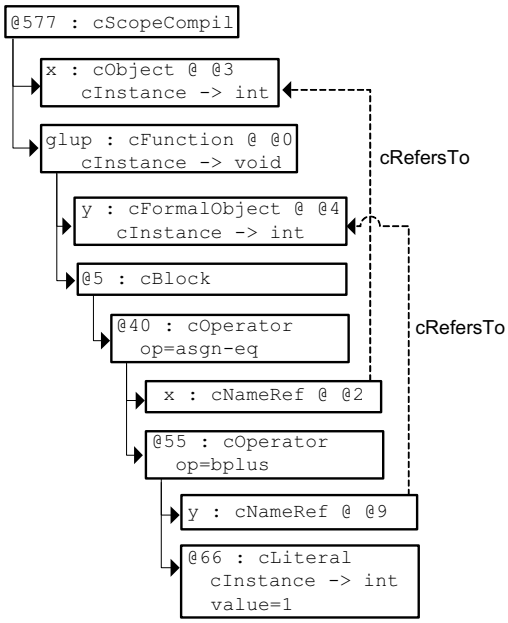

Figure 5. TA from example C program as a diagram

6.2. Nested Syntax.

In the second step, the ASG TA is translated to nested syntax. This was done with the Grok relational calculator [15] [16]. Grok stores and manipulates a relational model using algebraic operators, and can read and write TA. In order to reveal syntactic structure, Grok's operation *showtree* outputs the relational model in a form in which tree edges are represented implicitly by nesting of braces, { and }. In the Datrix schema the tree edges are *contain* and *cInstance*. Our example represented in *showtree* format exhibits the tree structure once again represented by syntax, as seen here:

```
@577 : cScopeCompil
{  x : cObject @ @3
      ( cInstance -> int @ @10 )
   glup : cFunction @ @0
      ( cInstance -> void @ @7 )
   {  y_1 : cFormalObject @ @4
         ( cInstance -> int @ @10 )
      @5 : cBlock
      {  @40 : cOperator  { op=asgn-eq }
         {  x : cNameRef @ @2
               ( cRefersTo -> x @ @3 )
            @55 : cOperator  { op=bplus }
            {  y_1 : cNameRef @ @9
               ( cRefersTo -> y_1 @ @4 )
            @66 : cLiteral
               { value=1 }
               ( cInstance -> int @ @10 )
            }
         }
      }
   }
}
```

6.3. TXL scripts.

The final step of RCPPX is carried out by two sets of TXL scripts [7]. TXL is a source transformation system based on context-free structure of input and output.

The first TXL script reads its input in the *showtree* format mentioned above. This format is structurally similar to the original program (TXL understands programs), rather than to the fact base (TXL does not understand data bases). The key idea is to use syntax (relative position in the token stream) to represent structural information, instead of using explicit relational edges.

TXL is well suited to translation from input according to one grammar into output according to a second grammar. TXL first inputs a *union grammar* [9] that encompasses the grammars of both the input and the output languages. Driven by a command script, the TXL processor repeatedly manipulates the parse tree of the input, until the tree assumes the form specified by the output grammar. Then TXL outputs the result as a source program, which satisfies the output grammar.

RCPPX drives TXL with a first set of scripts, which start by reading the ASG as represented in *showtree* format. These scripts carry out a pattern of simple *local rewrite* transformations in which subtrees are locally manipulated. For example, an input *showtree* expression of the form

```
cOperator { op = bplus } { E1 E2 }
```

is transformed to this C/C++ expression

```
(E1 + E2)
```

Here is the TXL rule that carries out the transformation:

```
replace [operator]
   cOperator { op = bplus }
         { E1 [expression] E2 [expression] }
by
   (E1 + E2)
```

This replaces each `operator` in the program of the form

```
cOperator { op = bplus } { E1 E2 }
```

with

```
(E1 + E2)
```

Next, RCPPX uses a second set of TXL scripts to carry out *local rotation* transformations, which change the order of entities. For example, an input showtree fragment of the form

```
p : cObject
{  cPtrType
```


IEEE COMPUTER SOCIETY

```
      {  int : cBuiltInType
      }
   }
```

is transformed to a C/C++ fragment of the form

```
      int *p;
```

This replaces *cPtrType* by a star (*) and "rotates" the fragment so that type `int` precedes the star and the star precedes identifier `p`. As well, the transformation deletes unwanted punctuation and keywords, and adds a semicolon. Here is the TXL rule that does the replacement and the renaming:

```
replace [object]
   cObject  cPtrType
      { T [type] }
      D [declarator]
construct ND [declarator]
   * D
by
   cObject  T ND
```

In this rule, `T` corresponds to a type, such as `int`, and `D` corresponds to an identifier, such as `p`. The rule constructs ND as a temporary value consisting of a star preceding `D`. The result of the transformation is

```
cObject T ND
```

whose value is

```
cObject T * D
```

This rule is executed repeatedly and recursively to replace each cPtrType by a star and to rearrange terms..

The transformations that rewrite the ASG in TA and *showtree* notation to source programs in C/C++ notation could have been written in a language such as PERL or C. However, they are much easier to write and debug in a special syntactic transformation language such as TXL. Since they are straightforward, and are done in very high level notations, we have confidence that they are carried out reliably.

7. Errors Found in CPPX

Before we developed the validation method described in this paper, CPPX had been used to extract facts from a number of large systems, including PostgreSQL, which consists of about 400,000 lines of source code. Because of this application of CPPX we knew that it was useable, but also that it had various bugs. This knowledge motivated us to develop the validation method described in this paper.

Here are three examples of the kinds of bugs we found in CPPX using our validation method.

1) Headers of for loops. A *for* loop header, e.g.,

```
for (i = 1; i < 10; i ++)
```

has, at most, three fields. For example, the third field is omitted here:

```
for (i = 1; i < 10)
```

We found that when CPPX produced facts for a header with omitted fields, these facts did not specify which fields were missing.

2) Details about data types. We found that CPPX did not correctly handle certain details about data types such as *struct* and *enum* type.

3) Missing identifiers. We found that CPPX sometimes omitted certain identifiers, such as names of structs.

We expect to find and correct more bugs in CPPX as our test suite grows and becomes more comprehensive.

8. Conclusions and Future Work

This paper presents a hierarchy of completeness for fact extractors and introduces the concept of relative completeness of extractors. It gives a new method for validating a fact extractor. The method avoids manual checking of generated fact bases. It works by recovering a version of the source program from the extracted factbase and compiling that version.

Given certain properties of a compiler (idempotent front end and deterministic back end), the method is able to validate semantic completeness of the extractor by checking that the generated assembly language is identical for the original and recovered source programs. This method was applied to the CPPX extractor.

The correctness of name resolution in the factbase was checked by the artifice of suffixing a number (a node key) to local identifiers.

In our application of this method to CPPX, we created a recovery transformation RCPPX. We did this using high level tools including TXL and Grok, so RCPPX is small and sufficiently reliable.

Our validation of CPPX has thus far been limited to source programs written in C; we expect to extend this work to C++ programs in the future. Our validation test suite has been limited to programs of modest size, to allow us to monitor results as needed. We plan to scale up these tests into a fully automatic testing framework.

We have an experimental Java extractor, but thus far it does not produce facts at the level of statements and expressions. When it is extended to extract these facts and to be semantically complete, we will be able to apply our validation method.

Our work involved two major surprises. First, we were surprised to discover that the assembly code for an original source program and for the version of the source program recovered from its extracted factbase were identical. Second, we were surprised to find a mechanical test for a special case of a generally undecidable problem. In particular, our method tests the semantic equivalence of two programs, by using properties of the programs and the compiler. Combining these two surprises, we developed a

IEEE COMPUTER SOCIETY

method for an automatic validation for semantic completeness of a fact extractor.

References

[1] A. Aho *et al*. *Principles of Compiler Design*. Addison-Wesley, 1977.

[2] A. Aho et al. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[3] A. Appel *Modern Compiler Implementation in Java*. Cambridge University Press. 1998.

[4] M.N. Armstrong et al. "Evaluating Architectural Extractors". In Fifth Working Conference on Reverse Engineering (WCRE '98) pp. 30-39, October 1998.

[5] Bell Canada, DATRIX™ Abstract Semantic Graph: Reference Manual, Version 1.4, Bell Canada Inc., Montreal, 2000.

[6] B. Bellay, H. Gall. "A Comparison of Four Reverse Engineering Tools". In Fourth Working Conference on Reverse Engineering (WCRE '97). pp. 2-11, Amsterdam. October 1997.

[7] J.R. Cordy *et al*. "Source Transformation in Software Engineering using the TXL Transformation System". In *Special Issue on Source Code Analysis and Manipulation, Journal of Information and Software Technology 44,13* (October 2002), pp. 827-837.

[8] T. R. Dean *et al.* "Union Schemas as a Basis for a C++ Extractor", In *Proceedings of WCRE 2001: Working Conference on Reverse Engineering*, Stuttgart, Germany, Oct 2-5, 2001

[9] T. R. Dean *et al*. "Grammar programming in TXL". In *Proceedings of Second IEEE International Workshop on Source Code Analysis and Manipulation*. Montréal, October 2002.

[10] J. Ebert *et al*. "Graph Based Modeling and Implementation with EER/GRAL". In Thalheim, B. *15th International Conference on Conceptual Modeling (ER'96), Proceedings*. LNCS 1157, pp. 163-178, Berlin. Springer-Verlag.

[11] Ferenc R. *et al*. "Columbus - Reverse Engineering Tool and Schema for C++" In *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, IEEE Computer Society, 2002

[12] P. J. Finnegan et al "The software bookshelf". *IBM Systems Journal, Vol. 36, No. 4*, pp. 564-593, November 1997. See http://www.research.ibm.com/journal/sj/ for the paper and http://swag.uwaterloo.ca/pbs/ for PBS.

[13] Free Software Foundation. *The GNU Compiler Collection*. See http://gcc.gnu.org/.

[14] R. C. Holt. "An Introduction to TA: the Tuple-Attribute Language, March 1997 (updated July 2002)." See http://plg.uwaterloo.ca/~holt/papers/ta-intro.htm

[15] R. C. Holt. "Introduction to the Grok Programming Language". From http://plg.uwaterloo.ca/~holt/papers/grok-intro.doc

[16] R. C. Holt. "Structural Manipulations of Software Architecture using Tarski Relational Algebra". In *WCRE '98: Working Conference on Reverse Engineering,* Honolulu, Oct 1998.

[17] T. C. Lethbridge et al. "The Dagstuhl Middle Model (DMM) Version 0.005 – Feb 20, 2002". See http://scgwiki.iam.unibe.ch:8080/Exchange/2.

[18] G. C. Murphy et al. "An Empirical Study of Static Call Graph Extractors". In ACM Transactions on Software Engineering and Methodology, 7(2): pp. 158-191, April 1998

[19] S. E. Sim et al. "On Using a Benchmark to Evaluate C++ Extractor", In Proc. IWPC '02, Paris, June, 2002

[20] A. Winter. "Exchanging Graphs with GXL". In P. Mutzel (ed.) *Graph Drawing - 9th Interational Symposium, GD 2001*, Vienna, Springer-Verlag. 2001.

[21] K. Wong. *The Rigi User's Manual - Version 5.4.4*. The Rigi Group, June 1998. See http://www.rigi.csc.uvic.ca/